

**UNIVERSITY OF MISSOURI-ST.LOUIS**

**DEPARTMENT OF COMPUTER SCIENCE**

A Project Report on  
**Natural Language Processing (NLP)**

**Submitted by:**

Sachina Koirala  
Student ID: 18281333

**Submitted to:**

**Dr. Badri Adhikari**

October 22, 2024

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Background</b>	<b>2</b>
<b>3</b>	<b>Problem Statement</b>	<b>2</b>
<b>4</b>	<b>Code Implementation</b>	<b>3</b>
<b>5</b>	<b>Method 1: Word Frequency-Based approach</b>	<b>3</b>
5.1	Steps Involved . . . . .	3
5.2	Results: . . . . .	5
5.3	Limitations . . . . .	6
5.4	Practical Implementation and Challenges . . . . .	6
<b>6</b>	<b>Method 2: TF-IDF with Cosine Similarity (More Accurate Approach)</b>	<b>7</b>
6.1	Steps Involved . . . . .	7
6.2	Improvements Over Method 1 . . . . .	7
6.3	Code Implementation for Method 2 . . . . .	7
6.4	Results: . . . . .	9
<b>7</b>	<b>Method 3: Hashing Vectorizer (Faster Approach)</b>	<b>9</b>
7.1	Steps Involved . . . . .	9
7.2	Improvements . . . . .	9
7.3	Code Implementation for method 3, using hashinhg vectorizer . . . . .	9
7.4	Results: . . . . .	11
<b>8</b>	<b>Challenges Faced</b>	<b>11</b>
<b>9</b>	<b>Conclusion and Discussion</b>	<b>12</b>
<b>10</b>	<b>References</b>	<b>12</b>

# 1 Introduction

Natural Language Processing (NLP) is a field of artificial intelligence that focuses on the interaction between computers and human languages. One of the major challenges in NLP is to develop models that can accurately measure the similarity between different pieces of text. The ability to quantify text similarity has numerous applications, including plagiarism detection, search engines, and text classification.

The purpose of this project is to explore various methods for determining text similarity using NLP techniques. Text similarity is a key problem in many areas of NLP, such as information retrieval, plagiarism detection, and automated essay scoring. We aim to implement, analyze, and compare multiple methods for computing text similarity within a dataset of essays, highlighting each method's strengths and limitations.

## 2 Background

There are multiple approaches to computing text similarity, ranging from simple word frequency-based methods to more advanced vectorization techniques and deep learning models. Word frequency-based approaches involve counting the occurrences of words in the text and comparing those counts to determine similarity. Although these methods are easy to implement, they often fail to capture the meaning behind the words, which can lead to inaccurate similarity measures. More sophisticated techniques, such as TF-IDF vectorization, are better equipped to capture the distinction of language and provide more accurate similarity measurements.

In this project, we will begin by implementing a basic word frequency-based approach and then explore more advanced techniques that aim to enhance the accuracy and efficiency of similarity computation.

## 3 Problem Statement

The objective of this project is to develop and implement different methods for measuring text similarity between pairs of essays. The specific goals are:

- To implement a word frequency-based method to measure the similarity between essays.
- To analyze the limitations of this method and identify cases where it produces misleading results.
- To implement two additional methods that either provide greater accuracy or improved computational efficiency compared to the word frequency-based approach.
- To compare the results across these different methods and evaluate their performance in terms of accuracy, speed, and practical usability.

The dataset used for this analysis consists of essays having total of over 17,307 essays.

## 4 Code Implementation

The following tools were used for the code implementation:

- Visual Studio Code
- Google Collaboratory
- Programming Language: Python
- Google Sheet: data-train.csv
- Copyleaks.com

The code implementation includes three methods for calculating text similarity: Method 1, Method 2, and Method 3.

## 5 Method 1: Word Frequency-Based approach

The first method focuses on calculating similarities between essays using word frequency vectors. To achieve this, **CountVectorizer** is used to convert the essays into word frequency vectors, followed by calculating cosine similarity between the resulting vectors.

### 5.1 Steps Involved

- **Preprocessing:** Tokenized and filtered essays containing "PROPER\_NAME" and removed duplicates.

```
from google.colab import drive
drive.mount('/content/drive')
import pandas as pd

# Load the dataset
train_data_path = '/content/drive/My Drive/data-train.csv'
train_df = pd.read_csv(train_data_path)

# Count the number of essays containing "PROPER_NAME"
num_essays_with_proper_name = train_df[train_df['full_text'].str.contains("PROPER_NAME", na=False)].shape[0]

# Count the number of duplicate essays based on 'full_text'
num_duplicates = train_df.duplicated(subset='full_text').sum()

# Remove essays containing "PROPER_NAME" and remove duplicates
filtered_df = train_df[~train_df['full_text'].str.contains("PROPER_NAME", na=False)].drop_duplicates(subset='full_text')

# Count the number of essays after filtering
```

```

num_essays_after_filtering = filtered_df.shape[0]

# Output the results
print(f"Number of essays containing 'PROPER_NAME': {
    num_essays_with_proper_name}")
print(f"Number of duplicate essays: {num_duplicates}")
print(f"Number of essays after filtering: {
    num_essays_after_filtering}")

```

Output of the above code:

Number of essays containing 'PROPER\_NAME': 279

Number of duplicate essays: 0

Number of essays after filtering: 17028

- **Vectorization:** Used **CountVectorizer** to convert essays into word frequency vectors.
- **Cosine Similarity Calculation:** Calculated cosine similarity between the word frequency vectors using the following formula:

$$\text{cosinesimilarity}(A, B) = \frac{A \cdot B}{\|A\| \|B\|} \quad (1)$$

Below is the code implementation to convert each essay into a vector representing the word frequency and how cosine similarity between all the essays are calculated to identify the most similar pairs.

```

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# Use the sample essays
sample_size = min(10000, filtered_df.shape[0])
sampled_df = filtered_df.sample(n=sample_size, random_state
    =42).reset_index(drop=True)

# Extract the 'full_text' column from the sampled dataset
essays = sampled_df['full_text']

# Create vectors using CountVectorizer
vectorizer = CountVectorizer(max_features=10000)
essay_vectors = vectorizer.fit_transform(essays)

# Calculate cosine similarity for the sampled essays
print("Calculating pairwise cosine similarity for the sampled
    essays...")
similarity_matrix = cosine_similarity(essay_vectors)

```

- **Top Similar Pairs:** Extracted the top 10 essay pairs with the highest similarity scores, excluding exact duplicates.

Following is the code implementation that results in top most similar pair of essays out of all.

```
# Find the top 10 similar pairs (excluding self-pairs and
    duplicates)
num_top_pairs = 10
top_similar_pairs = []

for i in range(similarity_matrix.shape[0]):
    for j in range(i + 1, similarity_matrix.shape[1]):
        if similarity_matrix[i, j] < 1.0: # Exclude exact
            duplicates (score of 1.0)
            top_similar_pairs.append((i, j, similarity_matrix[i,
                j]))

# Sort the pairs by similarity score in descending order and get
    the top pairs
top_similar_pairs = sorted(top_similar_pairs, key=lambda x: -x
    [2])[:num_top_pairs]

# Print the top similar pairs with their essay IDs
print("\nTop Similar Pairs of Essays among 10000 samples:")
for idx, (i, j, similarity) in enumerate(top_similar_pairs):
    essay_id_1, essay_id_2 = sampled_df.loc[i, 'essay_id'],
        sampled_df.loc[j, 'essay_id']
    preview_1, preview_2 = sampled_df.loc[i, 'full_text'][:100],
        sampled_df.loc[j, 'full_text'][:100]
    print(f"Pair {idx + 1}: Essay IDs {essay_id_1} and {
        essay_id_2} with Similarity Score: {similarity:.4f}")
```

## 5.2 Results:

The top 10 most similar essays:

Calculating pairwise cosine similarity for the sampled essays...

Top Similar Pairs of Essays among 10000 samples:  
Pair 1: Essay IDs 29aa983 and 6d25307 with Similarity Score: 0.9583  
Pair 2: Essay IDs 84a1b1a and 66ee32e with Similarity Score: 0.9544  
Pair 3: Essay IDs 84a1b1a and 287ed5e with Similarity Score: 0.9542  
Pair 4: Essay IDs 77b1295 and 84a1b1a with Similarity Score: 0.9536  
Pair 5: Essay IDs 77b1295 and 66ee32e with Similarity Score: 0.9527  
Pair 6: Essay IDs ef95422 and ea57a9c with Similarity Score: 0.9526  
Pair 7: Essay IDs 5bcf9b0 and 84a1b1a with Similarity Score: 0.9510  
Pair 8: Essay IDs e026924 and ebe2ce0 with Similarity Score: 0.9504  
Pair 9: Essay IDs 99e37ba and ebe2ce0 with Similarity Score: 0.9500  
Pair 10: Essay IDs e4cde6a and 84a1b1a with Similarity Score: 0.9498

✓ 2m 55s completed at 9:34 PM

Figure 1: Output for top 10 essays having higher cosine similarity

## 5.3 Limitations

- **Loss of Context:** Word frequency-based approaches do not capture the semantic meaning of words, which can lead to incorrect similarity scores when different words are used to convey the same ideas.
- **Time Complexity:** Calculating similarity for all essay pairs has a time complexity of  $O(n^2)$ , which makes it computationally expensive for large datasets.
- **Misleading High Similarity Scores:** In this method, the similarity is calculated based on the overlap of common words, leading to certain limitations.

Let's take an example :

Pair 2: Essay IDs **84a1b1a** and **66ee32e** with Similarity Score: **0.9544**

The Copyleaks comparison shows that shared factual descriptions heavily influence the similarity score, even if the underlying arguments are different.

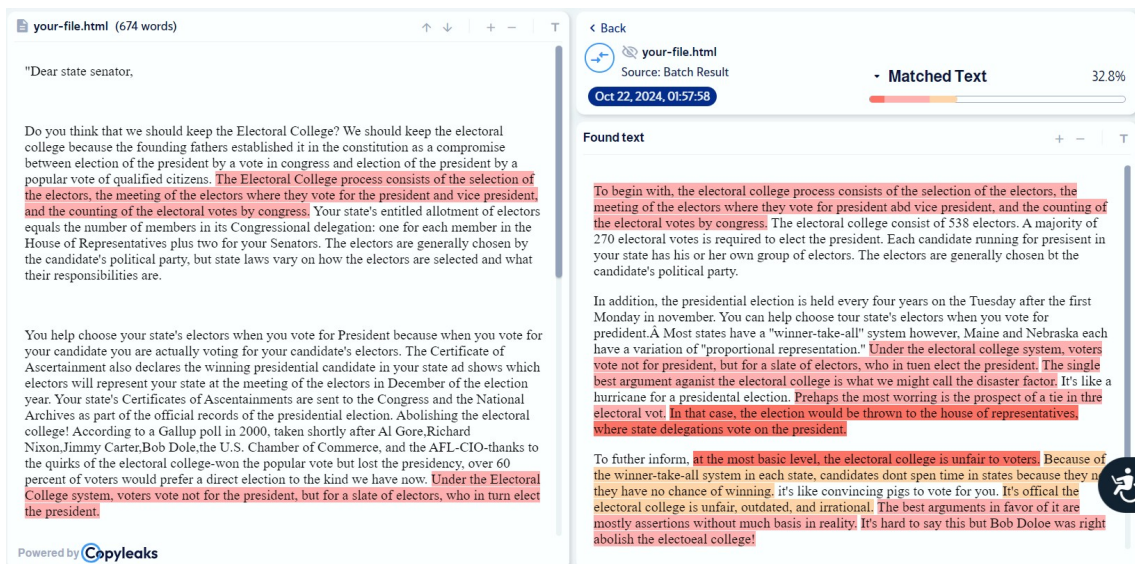


Figure 2: Comparison of pairs of essay having high similarity score

Therefore, cosine similarity can produce misleading high similarity scores, especially when essays share common definitions or core phrases about well-known topics like the Electoral College which serves as the major limitation of this method.

## 5.4 Practical Implementation and Challenges

Processing all 17028 essays at once led to system crashes due to memory limitations. Therefore, analysis on a subset of 10,000 essays is done for the code implementation.

## 6 Method 2: TF-IDF with Cosine Similarity (More Accurate Approach)

The second method involved using TF-IDF (Term Frequency-Inverse Document Frequency) to give more importance to informative words while down-weighting commonly occurring terms. This approach provides a more meaningful representation of essay content.

### 6.1 Steps Involved

- **Term Frequency (TF):** Calculated the frequency of words within each essay.
- **Inverse Document Frequency (IDF):** Measured how unique a word is across the entire dataset.
- **TF-IDF Score Calculation:** Calculated cosine similarity between TF-IDF vectors for all pairs of essays.

### 6.2 Improvements Over Method 1

- **Better Context Capture:** TF-IDF improved similarity scores by reducing the influence of common words and highlighting important terms.
- **Increased Accuracy:** Cosine similarity between TF-IDF vectors provided a better representation of essay content compared to word frequency vectors.
- **Scalability:** Unlike the word frequency method, TF-IDF allowed similarity calculations on all 17,028 essays without system crash issue.

### 6.3 Code Implementation for Method 2

First, import required libraries and remove the duplicates and calculate TF-IDF vectors as follows:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.neighbors import NearestNeighbors
import pandas as pd

# Remove essays containing "PROPER_NAME" and remove duplicates
print("Filtering the dataset...")
filtered_df = train_df[~train_df['full_text'].str.contains("
    PROPER_NAME", na=False)].drop_duplicates(subset='full_text')

# Set sampled_df to the entire filtered dataset instead of taking
# a sample
sampled_df = filtered_df.reset_index(drop=True)

essays = sampled_df['full_text']

# TF-IDF vectors
```



```
print("Creating TF-IDF vectors...")
vectorizer = TfidfVectorizer(max_features=10000)
tfidf_vectors = vectorizer.fit_transform(essays)
```

After that, calculate the cosine similarity: To find the most similar essays, the code uses NearestNeighbors from sklearn.neighbors, specifying cosine distance as the metric.

```
# Use NearestNeighbors to find the closest essays
print("Finding nearest neighbors using cosine similarity...")
num_neighbors = 11 # Include 10 nearest neighbors + the essay
                    itself
nbrs = NearestNeighbors(n_neighbors=num_neighbors, metric='cosine
                        ').fit(tfidf_vectors)

# Get the nearest neighbors (including distances)
distances, indices = nbrs.kneighbors(tfidf_vectors)

# Extract the top 10 similar pairs, excluding self-pairs (
  distance of 0)
top_similar_pairs = []

for i in range(distances.shape[0]):
    for j in range(1, num_neighbors):
        if distances[i, j] < 1.0:
            similarity_score = 1 - distances[i, j] # Convert
            cosine distance back to similarity
            top_similar_pairs.append((i, indices[i, j],
                                      similarity_score))

# Sort by similarity score in descending order and get the top 10
  pairs
num_top_pairs = 10
top_similar_pairs = sorted(top_similar_pairs, key=lambda x: -x
                           [2])[:num_top_pairs]

#Print the top similar pairs with their essay IDs
print("\nTop Similar Pairs of Essays (TF-IDF Cosine Similarity):"
      )
for idx, (i, j, similarity) in enumerate(top_similar_pairs):
    essay_id_1, essay_id_2 = sampled_df.loc[i, 'essay_id'],
    sampled_df.loc[j, 'essay_id']
    preview_1, preview_2 = sampled_df.loc[i, 'full_text'][:150],
    sampled_df.loc[j, 'full_text'][:150]
    print(f"Pair {idx + 1}: Essay IDs {essay_id_1} and {
          essay_id_2} with Similarity Score: {similarity:.4f}")
    print(f"Essay {essay_id_1} (Preview): {preview_1[:150]}...")
    print(f"Essay {essay_id_2} (Preview): {preview_2[:150]}...\n"
          )
```

## 6.4 Results:

```
Loading dataset...
Filtering the dataset...
Creating TF-IDF vectors...
Finding nearest neighbors using cosine similarity...

Top Similar Pairs of Essays (TF-IDF Cosine Similarity):
Pair 1: Essay IDs 29aa983 and 6d25307 with Similarity Score: 0.8872
Pair 2: Essay IDs 6d25307 and 29aa983 with Similarity Score: 0.8872
Pair 3: Essay IDs e026924 and 4d0c575 with Similarity Score: 0.8725
Pair 4: Essay IDs 4d0c575 and e026924 with Similarity Score: 0.8725
Pair 5: Essay IDs ef95422 and ea57a9c with Similarity Score: 0.8454
Pair 6: Essay IDs ea57a9c and ef95422 with Similarity Score: 0.8454
Pair 7: Essay IDs 84a1b1a and 287ed5e with Similarity Score: 0.8426
Pair 8: Essay IDs 287ed5e and 84a1b1a with Similarity Score: 0.8426
Pair 9: Essay IDs 5bcf9b0 and 287ed5e with Similarity Score: 0.8335
Pair 10: Essay IDs 287ed5e and 5bcf9b0 with Similarity Score: 0.8335
```

Figure 3: Output of similarity among the top similar pairs

## 7 Method 3: Hashing Vectorizer (Faster Approach)

The third method utilized the **HashingVectorizer**, which uses a hash function to map text tokens to a fixed-size feature space. This method aimed to provide a faster and more memory-efficient alternative to the previous method which is method1 .

### 7.1 Steps Involved

- **Hashing Trick:** Mapped words to a fixed-length vector using **HashingVectorizer**.
- **Cosine Similarity:** Calculated cosine similarity between hashed vectors.

### 7.2 Improvements

- **Memory Efficiency:** **HashingVectorizer** does not require storing a vocabulary, which reduces memory usage.
- **Faster Execution:** The fixed feature size enabled faster vectorization, making it suitable for larger datasets.

### 7.3 Code Implementation for method 3, using hashinhg vectorizer

In following code 10,000 essays were sampled similar to method 1 and HashingVectorizer was used to get the similar essays:

```
from sklearn.feature_extraction.text import HashingVectorizer
```

```

from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
import pandas as pd
import time

# Start measuring time
start_time = time.time()

# Remove essays containing "PROPER_NAME" and remove duplicates
filtered_df = train_df[~train_df['full_text'].str.contains("
    PROPER_NAME", na=False)].drop_duplicates(subset='full_text')

# Count the number of essays after filtering
num_essays_after_filtering = filtered_df.shape[0]

# Output the results
print(f"Number of essays after filtering: {
    num_essays_after_filtering}")

# Sample a subset of the essays
sample_size = min(10000, filtered_df.shape[0])
sampled_df = filtered_df.sample(n=sample_size, random_state=42).
    reset_index(drop=True)

# Extract the 'full_text' column from the sampled dataset
essays = sampled_df['full_text']

# Create vectors using HashingVectorizer
vectorizer = HashingVectorizer(n_features=10000, alternate_sign=
    False)
essay_vectors = vectorizer.fit_transform(essays)

# Calculate cosine similarity for the sampled essays
print("Calculating pairwise cosine similarity for the sampled
    essays using Hashing Vectorizer...")
similarity_matrix = cosine_similarity(essay_vectors)

# Find the top 10 similar pairs (excluding self-pairs and exact
    duplicates)
num_top_pairs = 10
top_similar_pairs = []

for i in range(similarity_matrix.shape[0]):
    for j in range(i + 1, similarity_matrix.shape[0]):
        if similarity_matrix[i, j] < 1.0: # Exclude exact
            duplicates (score of 1.0)
                top_similar_pairs.append((i, j, similarity_matrix[i,
                    j]))

# Sort the pairs by similarity score in descending order and get
    the top pairs

```

```

top_similar_pairs = sorted(top_similar_pairs, key=lambda x: -x
    [2])[:num_top_pairs]

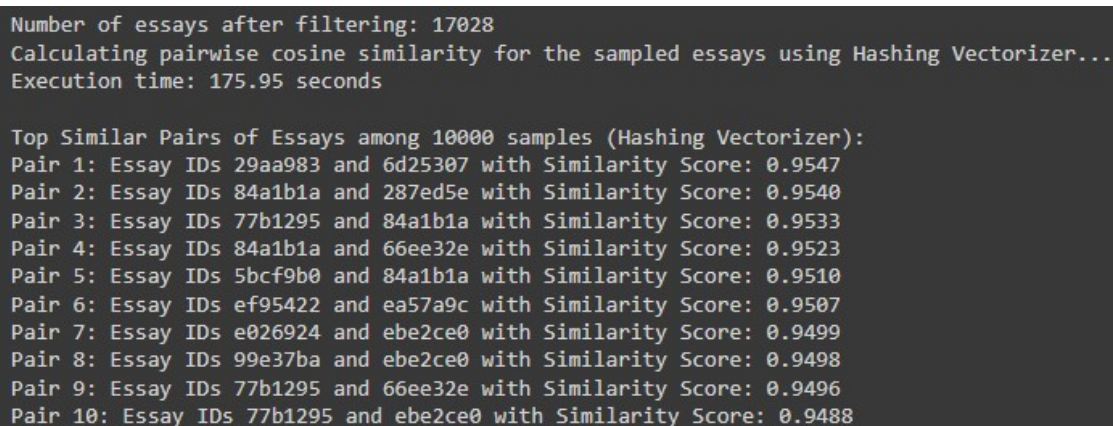
# Stop measuring time
end_time = time.time()
execution_time = end_time - start_time

# Print the execution time
print(f"Execution time: {execution_time:.2f} seconds")

# Print the top similar pairs with their essay IDs and one-line
# previews
print("\nTop Similar Pairs of Essays among 10000 samples (Hashing
    Vectorizer):")
for idx, (i, j, similarity) in enumerate(top_similar_pairs):
    essay_id_1, essay_id_2 = sampled_df.loc[i, 'essay_id'],
        sampled_df.loc[j, 'essay_id']
    preview_1, preview_2 = sampled_df.loc[i, 'full_text'][:100],
        sampled_df.loc[j, 'full_text'][:100]
    print(f"Pair {idx + 1}: Essay IDs {essay_id_1} and {
        essay_id_2} with Similarity Score: {similarity:.4f}")

```

## 7.4 Results:



```

Number of essays after filtering: 17028
Calculating pairwise cosine similarity for the sampled essays using Hashing Vectorizer...
Execution time: 175.95 seconds

Top Similar Pairs of Essays among 10000 samples (Hashing Vectorizer):
Pair 1: Essay IDs 29aa983 and 6d25307 with Similarity Score: 0.9547
Pair 2: Essay IDs 84a1b1a and 287ed5e with Similarity Score: 0.9540
Pair 3: Essay IDs 77b1295 and 84a1b1a with Similarity Score: 0.9533
Pair 4: Essay IDs 84a1b1a and 66ee32e with Similarity Score: 0.9523
Pair 5: Essay IDs 5bcf9b0 and 84a1b1a with Similarity Score: 0.9510
Pair 6: Essay IDs ef95422 and ea57a9c with Similarity Score: 0.9507
Pair 7: Essay IDs e026924 and ebe2ce0 with Similarity Score: 0.9499
Pair 8: Essay IDs 99e37ba and ebe2ce0 with Similarity Score: 0.9498
Pair 9: Essay IDs 77b1295 and 66ee32e with Similarity Score: 0.9496
Pair 10: Essay IDs 77b1295 and ebe2ce0 with Similarity Score: 0.9488

```

Figure 4: Output of similarity among pairs with execution time

## 8 Challenges Faced

- **System crashes:** Processing all essays caused frequent system crashes due to memory limitations. So, the sample size was reduced to 10,000 essays to prevent this in method 1 and method 3.
- **Manual adjustments:** Adjusting the dataset size to ensure successful execution while maintaining meaningful results was challenging.
- **High computation complexity:** For Method 1, computing cosine similarity be-

tween all essay pairs led to high computational costs which is ( $O(n^2)$ ) complexity. This made processing large datasets impractical without reducing the sample size.

- **Difficulty in visualization:** Visualizing similarity scores across all essay pairs was difficult, given the large number of comparisons using heatmap.
- **Accuracy vs Performance:** In Method 3 (Hashing Vectorizer Method), achieving a balance between speed and accuracy was challenging.

## 9 Conclusion and Discussion

- **Word frequency method:** Simple but limited in its ability to capture semantic meaning.
- **TF-IDF method:** Provided a more accurate representation of text, making it computationally feasible for the full dataset.
- **Hashing Vectorizer method:** Efficient for large datasets, but sacrifices interpretability.

The TF-IDF method proved to be the most accurate, while the Hashing Vectorizer offered the best efficiency. The choice of method depends on specific requirements, such as prioritizing accuracy or speed.

## 10 References

Paul Minogue. Available at: <https://paulminogue.com/index.php/2019/09/29/introduction-to-cosine-similarity/> (Accessed: 12 October 2024).

Liu, Z. et al. (2023) ‘Optimized algorithm design for text similarity detection based on Artificial Intelligence and Natural Language Processing’, *Procedia Computer Science*, 228, pp. 195–202. doi:10.1016/j.procs.2023.11.023.

Jain, A. (2024) TF-IDF vectorization with cosine similarity, Medium. Available at: <https://medium.com/@anurag-jain/tf-idf-vectorization-with-cosine-similarity-eca3386d4423> (Accessed: 15 October 2024).

Ganesan, K. (2020) Hashingvectorizer vs. Countvectorizer, Kavita Ganesan, PhD. Available at: <https://kavita-ganesan.com/hashingvectorizer-vs-countvectorizer/> (Accessed: October 2024).

Grammarly: An online tool that helps with grammar, punctuation, and style

CopyLeaks: An online tool to compare the essays and get the matching percentage